

4

# La abstracción procedimental

Grado en Ingeniería Electrónica de Comunicaciones

Luis Hernández Yáñez

Raquel Hervás Ballesteros

Virginia Francisco Gilmartín

Javier Arroyo Gallardo

Facultad de Informática  
Universidad Complutense



# Índice

---

Diseño descendente

Subprogramas

Funciones

Procedimientos

Datos en los programas

Variables globales

Variables locales

Parámetros

Por valor

Por referencia

Notificación de errores

Cuando usar subprogramas

Refinamientos sucesivos (un ejemplo)



# Diseño descendente (I)

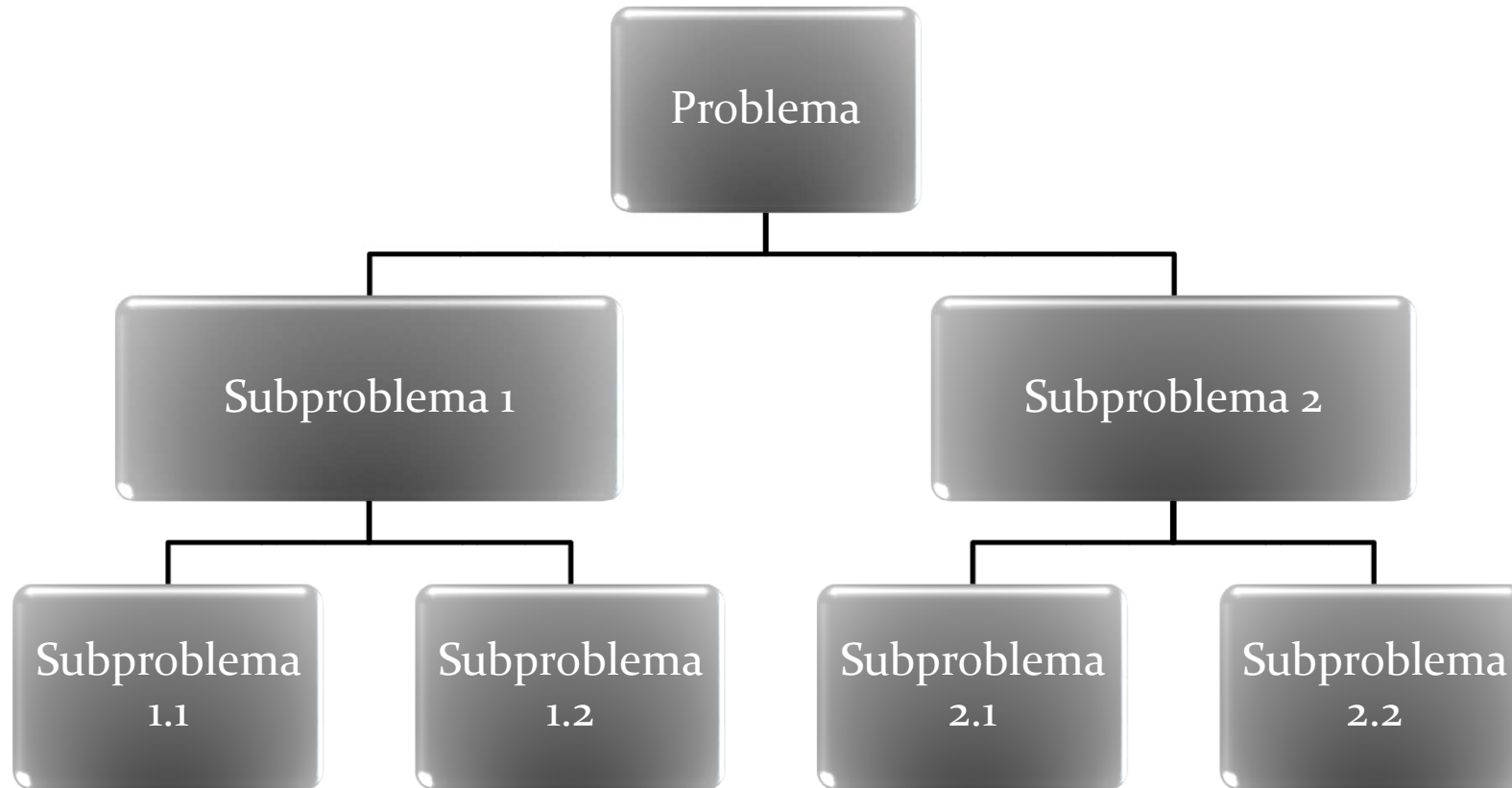
---

- Diseñar un algoritmo o un programa complejo de una sola vez es complicado
- El diseño descendente permite distinguir las distintas partes de un algoritmo o de un programa, resolverlas por separado y combinar las soluciones más pequeñas para obtener la solución final al algoritmo complejo
- **Diseño descendente o refinamiento por pasos sucesivos:** Desarrollo de un programa identificando primero grandes acciones y descendiendo a sus detalles progresivamente



# Diseño descendente (II)

---



# Diseño descendente (III)

---

Refinamientos sucesivos:

- ✓ Las tareas que ha de realizar un programa se pueden dividir en subtareas más sencillas
- ✓ Las subtareas también se pueden dividir en otras más sencillas
- ✓ ...

Cada tarea se descompone en subtareas en que se varía el nivel de detalle:

- ✓ Todas la subtareas deben estar al mismo nivel
- ✓ Cada subtaska debe poder ser abordada por separado
- ✓ La solución de cada subtaska debe poder unirse al resto para obtener la solución de la tarea



# Diseño descendente. Ejemplo Dibujo (I)



*Dibujar*

1. Dibujar

2. Dibujar

3. Dibujar

REFINAMIENTO

1. Dibujar

2. Dibujar

2.1. Dibujar

2.2. Dibujar

3. Dibujar

Misma tarea



# Diseño descendente. Ejemplo Dibujo (II)



1. Dibujar ○

2. Dibujar △

2.1. Dibujar ^

2.2. Dibujar —

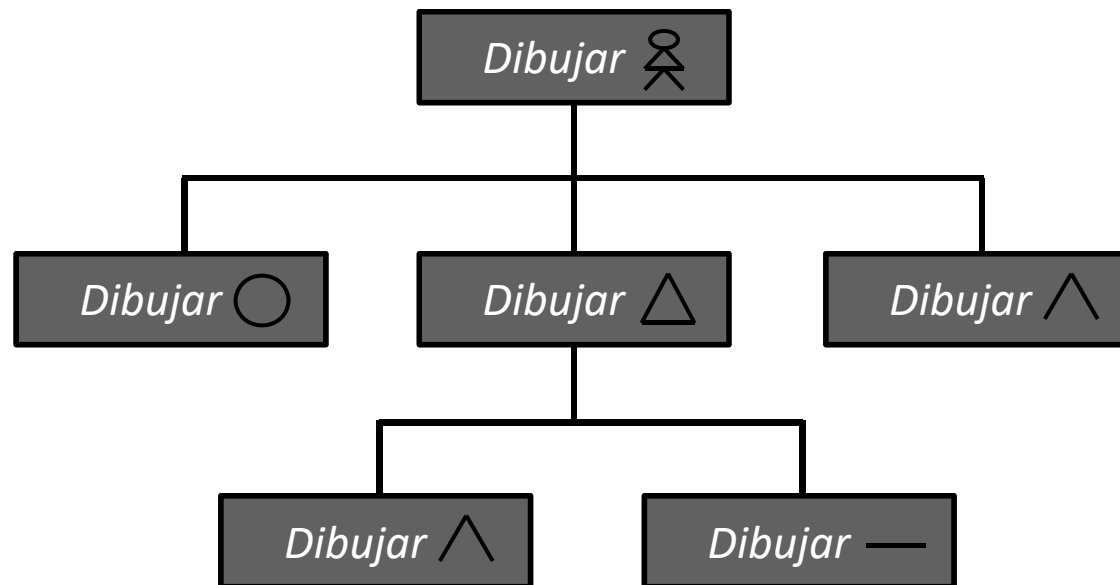
3. Dibujar ^

4 tareas, pero dos de ellas son iguales

Nos basta con saber cómo dibujar:



# Diseño descendente. Ejemplo Dibujo (III)



```
void dibujarCirculo()
{ ... }
```

```
void dibujarSecantes()
{ ... }
```

```
void dibujarLinea()
{ ... }
```

```
void dibujarTriangulo()
{
    dibujarSecantes();
    dibujarLinea();
}
```

```
int main() {
    dibujarCirculo();
    dibujarTriangulo();
    dibujarSecantes();
    return 0;
}
```





# Diseño descendente. Ejemplo Mensaje en letras gigantes (I)

Mostrar la cadena HOLA MAMA en letras gigantes

```
*   *  
*   *  
*****  
*   *  
*   *
```

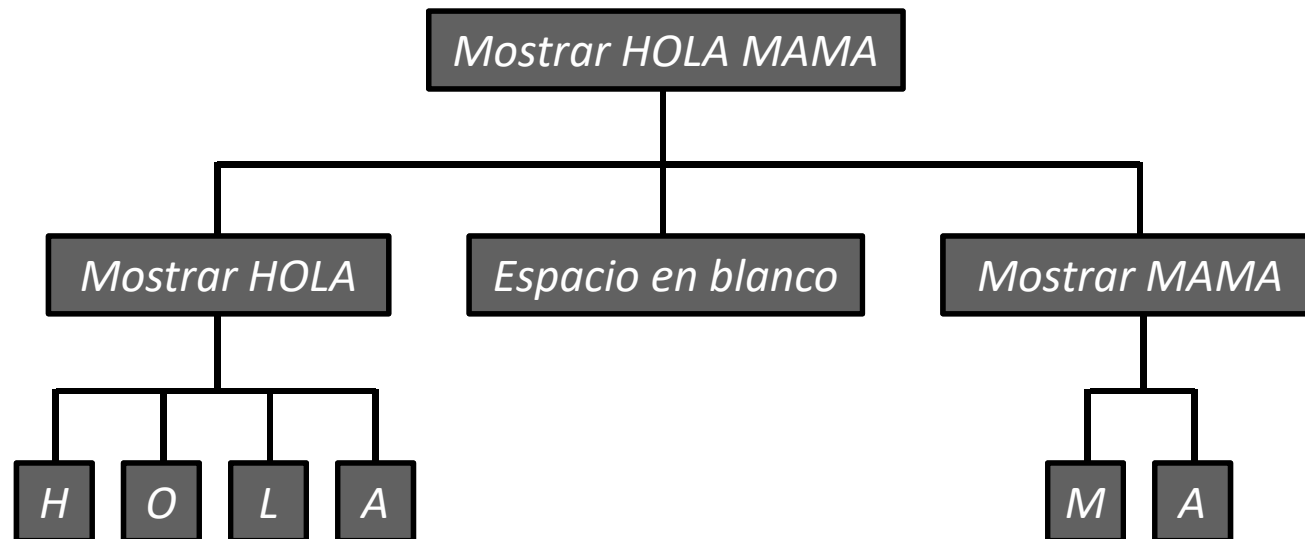
```
*****  
*   *  
*   *  
*   *  
*****
```

```
* * *
```



## Diseño descendente. Ejemplo Mensaje en letras gigantes (II)

Mostrar la cadena HOLA MAMA en letras gigantes



Tareas básicas



## Diseño descendente. Ejemplo Mensaje en letras gigantes (III)

---

```
void mostrarH() {  
    cout << "*" << endl;  
    cout << "*" << endl;  
    cout << "*****" << endl;  
    cout << "*" << endl;  
    cout << "*" << endl << endl;  
}
```

```
void mostrarO() {  
    cout << "*****" << endl;  
    cout << "*" << endl;  
    cout << "*" << endl;  
    cout << "*" << endl;  
    cout << "*****" << endl << endl;  
}
```

```
void mostrarL()  
{ ... }
```

```
void mostrarA()  
{ ... }
```

```
void espaciosEnBlanco() {  
    cout << endl << endl << endl;  
}
```

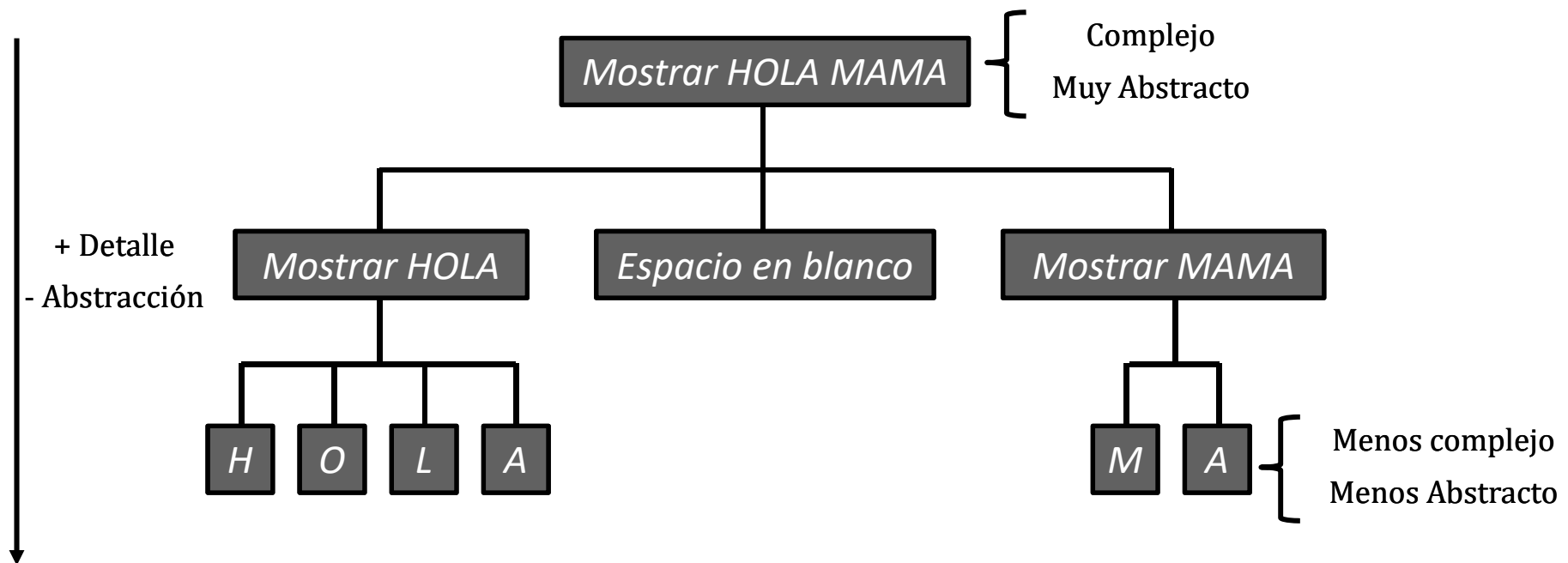
```
void mostrarM()  
{ ... }
```

```
int main() {  
    mostrarH();  
    mostrarO();  
    mostrarL();  
    mostrarA();  
    espaciosEnBlanco();  
    mostrarM();  
    mostrarA();  
    mostrarM();  
    mostrarA();  
  
    return 0;  
}
```



# Diseño descendente. Resolución de problemas

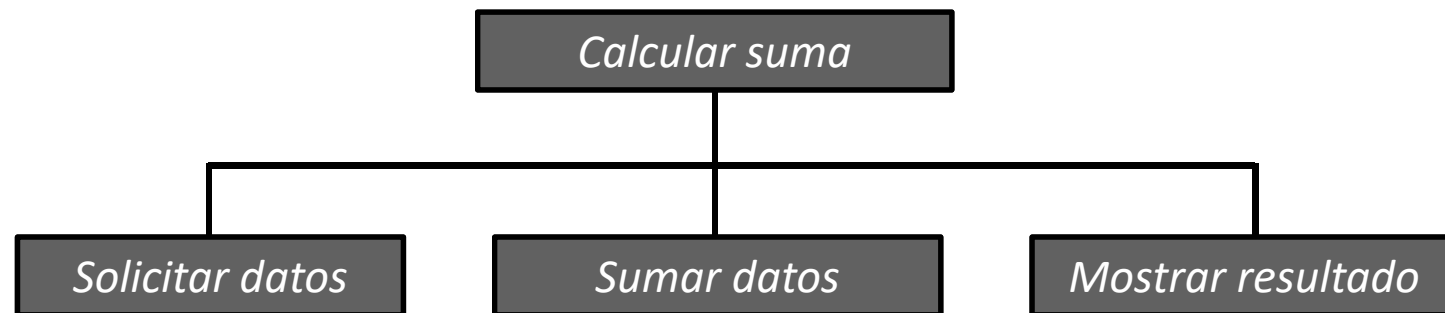
Descomponemos el problema en subproblemas cada vez más sencillos y concretos (disminuimos el nivel de abstracción)



# Diseño descendente. Resolución de programas

---

Supongamos que deseamos escribir un programa que calcule la suma de dos números naturales introducidos por el usuario y mostrarla por pantalla



La solución a cada subproblema será un subprograma



# Subprogramas

---

Pequeños programas dentro de otros programas

- ✓ Unidades de ejecución independientes
- ✓ Encapsulan código y datos
- ✓ Realizan tareas individuales del programa
- ✓ Funcionalidad concreta, identificable y coherente
- ✓ Se ejecutan de principio a fin cuando se llaman (*invocan*)
- ✓ Terminan devolviendo el control al punto de llamada
- ✓ Pueden ser utilizados tantas veces como se desee en el mismo programa o en otros programas



Aumentan el nivel de abstracción del programa  
Facilitan la prueba, la depuración y el mantenimiento



# Subprogramas. Ventajas

---

1. Mejoran la legibilidad del código
2. Acortan los programas
  - Evitan tener que escribir varias veces las mismas instrucciones
3. Permiten reutilizar el código
  - Los subprogramas se pueden reutilizar en otros programas acortando su tiempo de realización



# Subprogramas en C++

---

Forma general de un subprograma en C++:

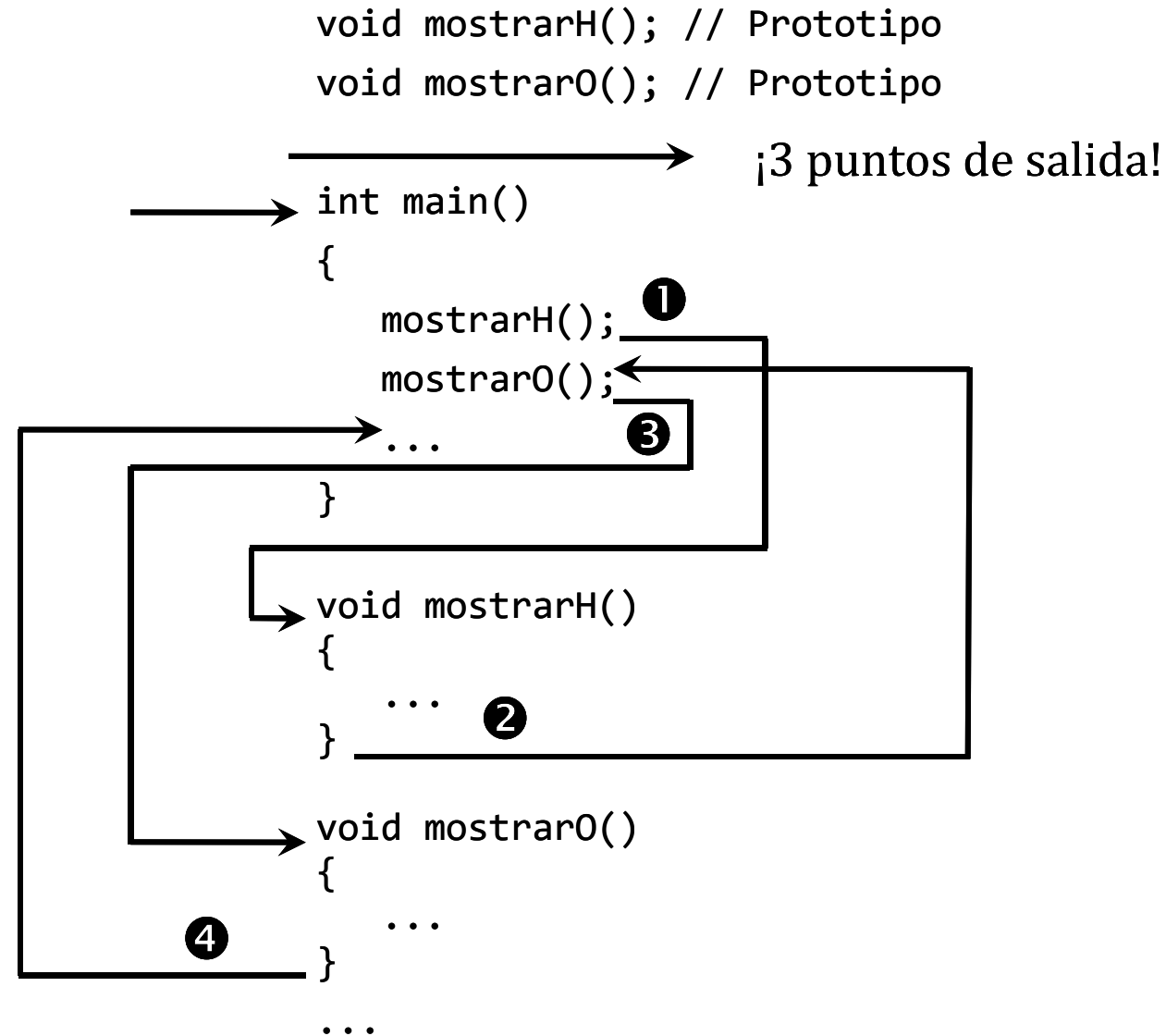
```
tipo nombre(parámetros) // Cabecera  
{  
    // Cuerpo  
}
```

- ✓ *Tipo* de dato que devuelve el subprograma como resultado
- ✓ *Parámetros* para la comunicación con el exterior
- ✓ *Cuerpo*: ¡Un bloque de código!





# Subprogramas. Flujo de ejecución



# Tipos de subprogramas

---

- ❖ Funciones: Son subprogramas que realizan una determinada tarea y devuelven un único resultado
- ❖ Procedimientos: Son subprogramas que realizan una determinada tarea y **no devuelven nada o devuelven más de un resultado**



# Funciones

---

- ✓ Devuelven un único resultado con la instrucción `return`
- ✓ Tipo distinto de `void`
- ✓ Llamada o invocación: en una asignación o dentro de cualquier expresión. Ejemplo:

```
y = raiz(144);
```

```
cout << 12 * y + cuadrado(20) - 3;
```

La llamada de la función se sustituye por el valor que devuelve al terminar.

- ✓ Los parámetros son solo de entrada (no se pueden modificar)



# Funciones. Ejemplo

---

```
...
int menu()
{
    int op;
    cout << "1 - Editar" << endl;
    cout << "2 - Combinar" << endl;
    cout << "3 - Publicar" << endl;
    cout << "0 - Cancelar" << endl;
    cout << "Elija: ";
    cin >> op;
    return op;
}

int main()
{
    ...
    int opcion;
    opcion = menu();
    ...
}
```

The diagram illustrates the execution flow between the `main` function and the `menu` function. An arrow points from the `menu()` call in `main` to the `menu` function definition. Another arrow points from the `return op;` statement in `menu` back to the assignment `opcion = menu();` in `main`, indicating the return of the value stored in `op`.



# Resultado de una función

## Una función ha de devolver un resultado

La función ha de terminar su ejecución devolviendo el resultado

instrucción `return` hace dos cosas:

- Devuelve el dato que se indica a continuación como resultado
- Termina la ejecución de la función

El dato devuelto sustituye a la llamada de la función en la expresión

```
int cuad(int x) {  
    return x * x;  
    x = x * x;  
}  
  
int main() {  
    cout << 2 * cuad(16);  
    return 0;  
}
```

Diagram illustrating function calls and return values:

- The function `cuad(16)` is called from `main()`.
- The return value of `cuad(16)` is `256`.
- The code between the `return` statement and the closing brace of `cuad` is never executed.



# Ejemplo: Cálculo del factorial

factorial.cpp

Factorial (N) = 1 x 2 x 3 x ... x (N-2) x (N-1) x N

```
long long int factorial(int n); // Prototipo
```




```
int main() {  
    int num;  
    cout << "Num: ";  
    cin >> num;  
    cout << "Factorial de " << num << ": " << factorial(num) << endl;  
    return 0;  
}
```

```
long long int factorial(int n) {  
    long long int fact = 1;  
    if (n < 0) {  
        fact = 0;  
    }  
    else {  
        for (int i = 1; i <= n; i++) {  
            fact = fact * i;  
        }  
    }  
    → return fact;  
}
```

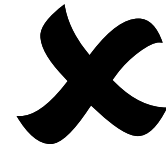


# Un único punto de salida (I)

---

```
int compara(int val1, int val2) {  
    // -1 si val1 < val2, 0 si iguales, +1 si val1 > val2  
    if (val1 == val2) {  
        return 0;   
    }  
    else if (val1 < val2) {  
        return -1;   
    }  
    else {  
        return 1;   
    }  
}
```

¡3 puntos de salida!





Para facilitar la depuración y el mantenimiento,  
codifica los subprogramas con un único punto de salida



# Un único punto de salida (II)

---

```
int compara(int val1, int val2) {  
    // -1 si val1 < val2, 0 si iguales, +1 si val1 > val2  
    int resultado;  
  
    if (val1 == val2) {  
        resultado = 0;  
    }  
    else if (val1 < val2) {  
        resultado = -1;  
    }  
    else {  
        resultado = 1;  
    }  
  
    return resultado;  Punto de salida único   
}
```



Para facilitar la depuración y el mantenimiento,  
codifica los subprogramas con un único punto de salida





# Procedimientos

---

- ✓ Se utilizan para estructurar un programa y mejorar su claridad
- ✓ Tipo: `void`
- ✓ No devuelven ningún resultado o devuelven más de un resultado
- ✓ No utilizan la palabra reservada `return`
- ✓ Llamada: instrucción independiente  
`mostrarH();`
- ✓ Los parámetros pueden ser:
  - De entrada: Sólo se utilizan para que los subprogramas que llaman al procedimiento le pasen datos al mismo
  - De salida: Sólo se utilizan para que el procedimiento pase los resultados obtenidos al exterior
  - De entrada/salida: Se utilizan por parte de los subprogramas que llaman, para pasarle datos al procedimiento, y por parte del procedimiento para pasar los resultados obtenidos al subprograma que lo ha llamado



# Procedimientos. Ejemplo

## Subprogramas de tipo void

```
...  
void menu()  
{  
    int op;  
    cout << "1 - Editar" << endl;  
    cout << "2 - Combinar" << endl;  
    cout << "0 - Cancelar" << endl;  
    cout << "Opción: ";  
    cin >> op;  
    if (op == 1) {  
        editar();  
    }  
    else if (op == 2) {  
        combinar();  
    }  
}
```

```
int main()  
{  
    ...  
    menu();  
    ...  
}
```



En el caso de los menús es preferible usar una función como en la transparencia anterior



# ¿Cuándo termina el subprograma?

---

Procedimientos (tipo `void`):

- Al encontrar la llave de cierre que termina el subprograma

Funciones (tipo distinto de `void`):

- Al encontrar una instrucción `return` (con resultado)



# Datos en los Programas

---

Existen 3 tipos de datos en los programas:

1. Variables globales: Son declaradas al principio del programa y son accesibles y visibles en todo el programa
2. Variables locales: Accesibles y visibles sólo dentro del bloque o subprograma donde están definidas
3. Parámetros declarados en la cabecera del subprograma: Comunican los subprogramas con el resto del programa. Los subprogramas reciben a través de los parámetros los valores con los que van a trabajar



# Variables globales (I)

---

- Son accesibles y visibles en todo el programa
- Se pueden usar en cualquier lugar (y subprograma) del programa
- Cualquier subprograma puede acceder a ellas y modificar su valor
- Son declaradas al principio del programa (antes del main)
- Tiempo de vida: toda la ejecución



# Variables globales (II)

---

## ➤ TERMINANTEMENTE PROHIBIDO UTILIZARLAS:

- Ocupan memoria durante toda la ejecución del programa
- Pueden ser modificadas por cualquier función
- Excepciones:
  - Constantes globales (valores inalterables que se usan en varios subprogramas)
  - Tipos globales (necesarios en varios subprogramas)
- ¿Necesidad de datos externos?
  - Define parámetros en el subprograma
  - Los datos externos se pasan como argumentos en la llamada



# Variables locales

---

- Están definidas dentro de un subprograma o un bloque de código
- No pueden usarse fuera del subprograma o bloque de código donde están definidas
- Solo ocupan memoria mientras existen:
  - Se crean cuando se ejecuta el subprograma o bloque donde están definidas
  - Se destruyen al salir del subprograma o bloque donde están definidas
- Se pueden declarar variables en cualquier parte del código pero no se pueden usar hasta que no se declaren
- Datos locales a un bloque ocultan otros externos homónimos



# Datos locales y datos globales. Ejemplo

```
#include <iostream>
using namespace std;
```

```
const int MAX = 100;
double ingresos;
```

} Datos globales

op de proc() es distinta de op de main()

```
...
void proc() {
    int op;
    double ingresos;
```

} Datos locales a proc()

... → Se conocen MAX (global), op (local) e ingresos (local que oculta la global)

```
int main() {
    int op;
```

} Datos locales a main()

... → Se conocen MAX (global), op (local) e ingresos (global)

```
    return 0;
}
```





# Parámetros

---

- Los parámetros son los valores que se pasan a los subprogramas al llamarlos
- Dentro del subprograma se tratan igual que una variable local:
  - Se crean al comenzar la ejecución del subprograma
  - Se destruyen al terminar la ejecución del subprograma
- En la llamada o invocación al subprograma los argumentos deben de coincidir en número y tipo con los parámetros
  - No es necesario que coincidan en nombre
  - Cada parámetro es sustituido por el argumento que en la llamada al subprograma ocupa la misma posición



# Parámetros en C++

## *Declaración de parámetros*

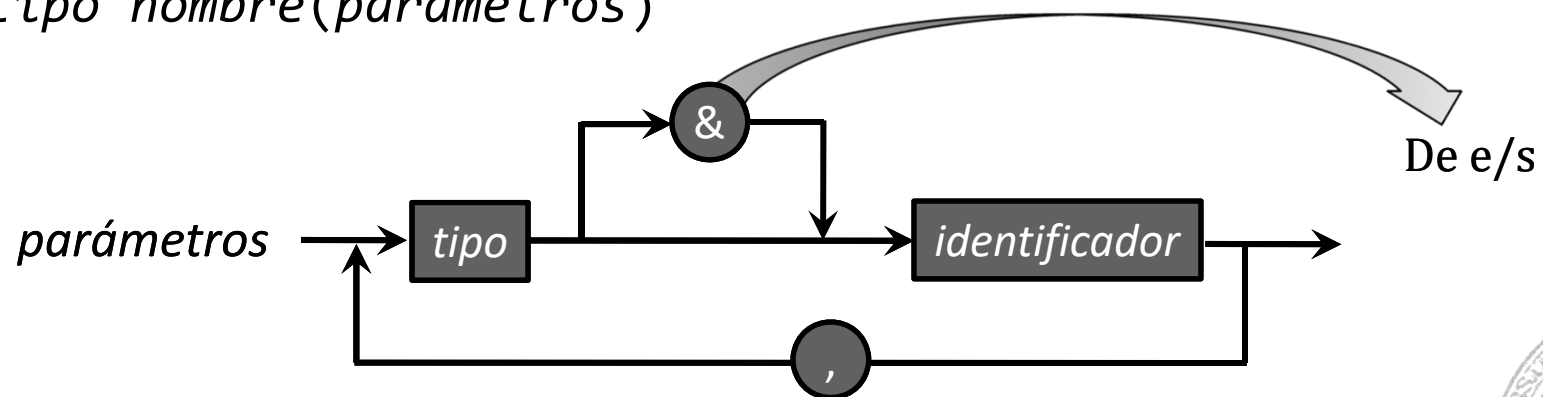
Sólo dos clases de parámetros en C++:

- Sólo de entrada (*por valor*)
- De entrada / salida (*por referencia*)

## *Lista de parámetros*

Entre los paréntesis de la cabecera del subprograma

*tipo nombre(parámetros)*

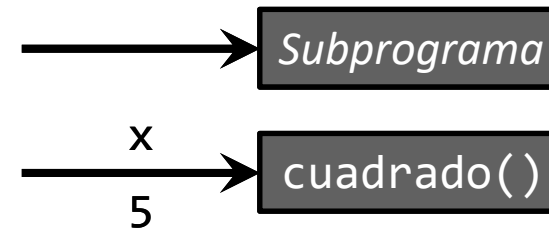


# Comunicación con el exterior

Datos de entrada, datos de salida y datos de entrada/salida

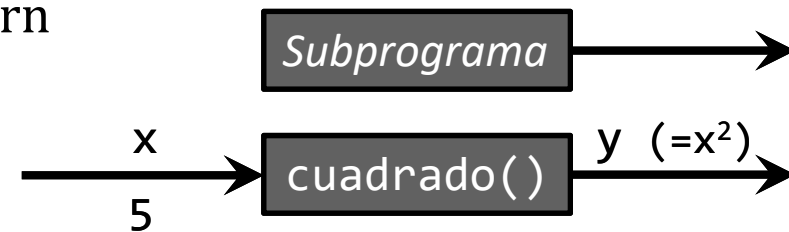
✓ Datos de entrada: Aceptados

—Subprograma que dado un número muestra en la pantalla su cuadrado:



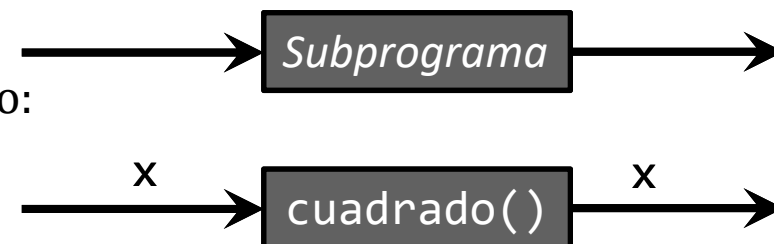
✓ Datos de salida: Devueltos con return

—Subprograma que dado un número devuelve su cuadrado:



✓ Datos de entrada/salida: Aceptados y modificados

—Subprograma que dada una variable numérica la eleva al cuadrado:



# Parámetros por valor (I)

---

- ✓ Son parámetros solo de entrada
- ✓ Se usan para pasar datos al subprograma
- ✓ Al subprograma se le pasa una **COPIA** del argumento

`int cuadrado(int num)`

`double potencia(double base, int exp)`

`void muestra(string nombre, int edad, string nif)`

`void proc(char c, int x, double a, bool b)`

- ✓ Argumentos: Expresiones en general: variables, constantes, literales, llamadas a función, operaciones,...
- ✓ Se destruyen al terminar la ejecución del subprograma
- ✓ Cualquier modificación sufrida por el parámetro dentro de la función no afectará al argumento, al finalizar el subprograma el valor del argumento no ha cambiado



# Parámetros por valor (II)

---

✓ Por defecto, en C++ todo paso de parámetros se hace por valor, excepto el paso por parámetro de los arrays que por defecto se hace por referencia

✓ Para pasar los arrays por valor hay que pasarlos como constantes (lo veremos en su momento):

```
double media(const TArray lista)
```

✓ Todos los ejemplos de definición de funciones y llamadas a funciones vistas hasta este tema implementan el paso de parámetros por valor



# Argumentos pasados por valor (I)

---

Expresiones válidas con concordancia de tipo:

`void proc(int x, double a) → proc(23 * 4 / 7, 13.5);`

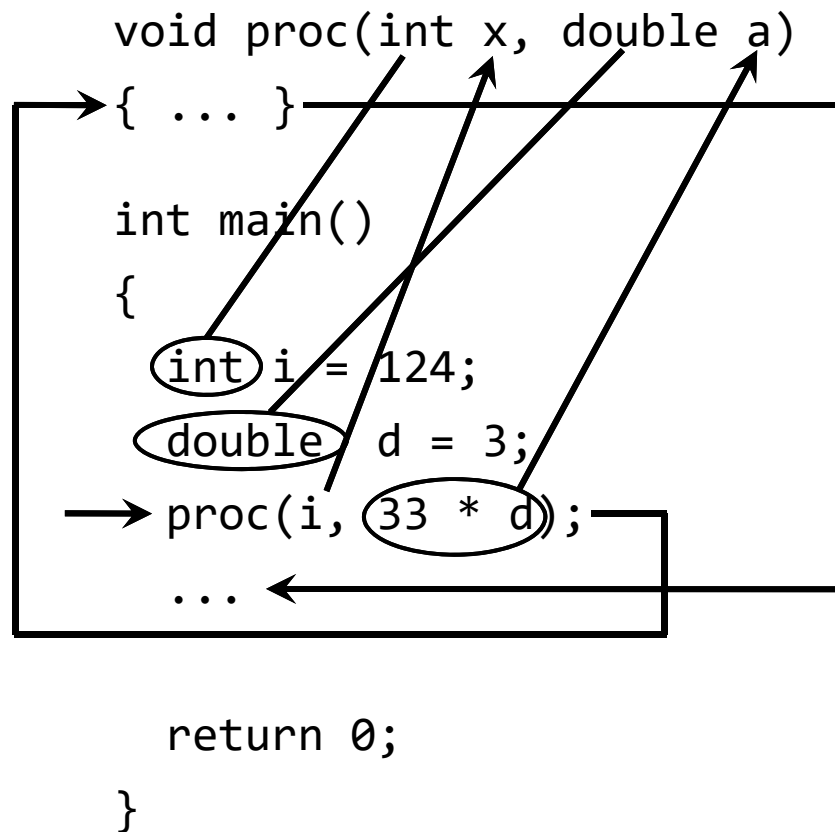
→ `double d = 3;  
proc(12, d);`

→ `double d = 3;  
int i = 124;  
proc(i, 33 * d);`

→ `double d = 3;  
int i = 124;  
proc(cuad(20) * 34 + i, i * d);`



# Argumentos pasados por valor (II)



Memoria	
i	124
d	3.0
...	
...	
x	124
a	99.0
...	



# Parámetros por referencia (I)



- ✓ Son parámetros de Salida o de Entrada/Salida
- ✓ Se usan para pasar datos al subprograma y para que el subprograma devuelva los datos obtenidos
- ✓ Al subprograma se le pasa la dirección de memoria del argumento
- ✓ Para pasar un parámetro por referencia se debe poner & (operador de dirección). Ejemplos:

```
void incrementa(int &x)
```

```
void intercambia(double &x, double &y)
```

```
void proc(char &c, int &x, double &a, bool &b)
```

- ✓ Solo se pueden pasar por referencia variables (no se pueden pasar expresiones)
- ✓ El paso por referencia permite que un subprograma pueda devolver más de un valor (las funciones solo pueden devolver un valor)





# Parámetros por referencia (II)



- ✓ Los argumentos pueden quedar modificados. Cualquier modificación en el parámetro afecta al argumento.
  - El subprograma trabaja con el propio argumento y no con la copia como ocurría en los parámetros por valor
- ✓ *¡No usaremos parámetros por referencia en las funciones!*  
Sólo en procedimientos



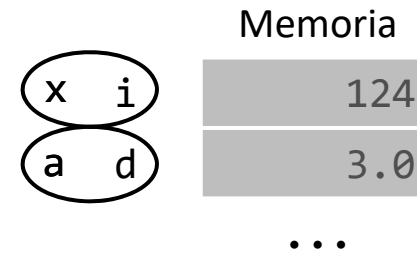
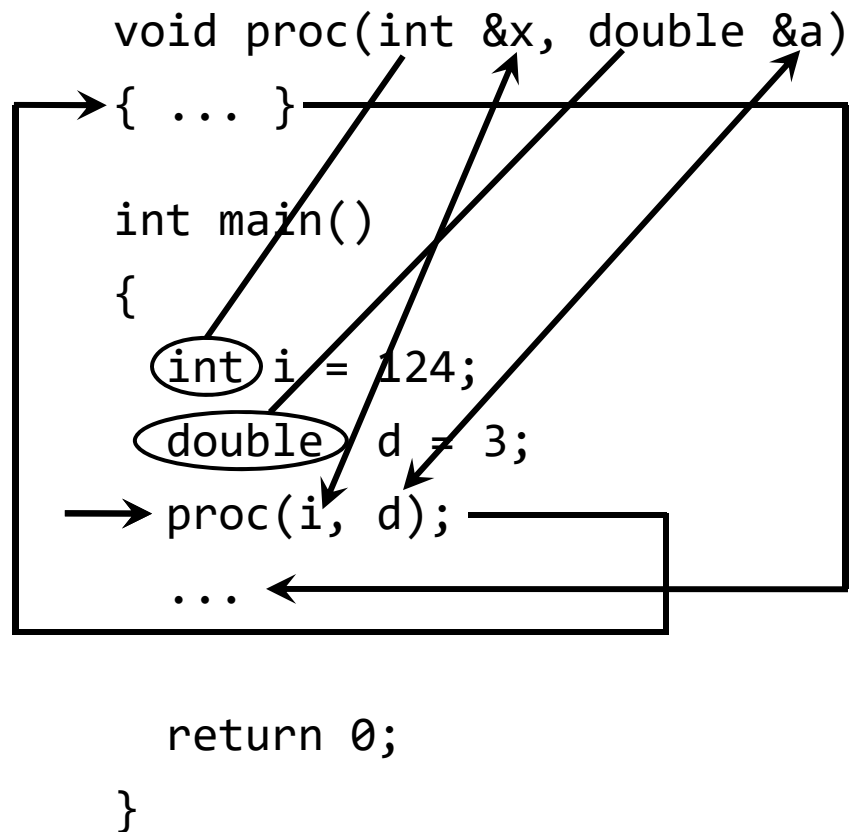
En un subprograma puede haber tanto por valor como por referencia

*¡Atención!* Los arrays se pasan por referencia sin utilizar &  
`void insertar(tArray lista, int &contador,  
double item)`

El argumento `lista` (variable de tipo `tArray`) quedará modificado



# Argumentos pasados por referencia



# Tipos de parámetros

---

- Parámetros de Entrada:
  - ✓ Este tipo de parámetros llegan del exterior y no pueden ser modificados en el subprograma
    - No deben de aparecer NUNCA en la parte izquierda de una asignación dentro del subprograma
  - ✓ Sin & y arrays con const
- Parámetros de Salida:
  - ✓ El valor no llega desde el exterior sino que se genera dentro del subprograma y se saca fuera
    - No debe aparecer NUNCA en la parte derecha de una asignación dentro del subprograma, solo a la izquierda
  - ✓ Con & (excepto los arrays que van sin &)
- Parámetros de Entrada y Salida:
  - ✓ El valor llega desde el exterior y puede ser modificado en el subprograma
    - Puede aparecer tanto a la derecha como a la izquierda de una asignación dentro del subprograma
  - ✓ Cualquier cambio en el parámetro se ve reflejado fuera del subprograma
  - ✓ Con & (excepto los arrays que van sin &)



# ¿Qué llamadas son correctas?

---

Dadas las siguientes declaraciones:

```
int i;
```

```
double d;
```

```
void proc(int x, double &a);
```

*¿Qué pasos de argumentos son correctos? ¿Por qué no?*

<code>proc(3, i, d);</code>	<b>✗</b>	Nº de argumentos ≠ Nº de parámetros
<code>proc(i, d + 1);</code>	<b>✗</b>	Parámetro por referencia → ¡variable!
<code>proc(3 * i + 12, d);</code>	<b>✓</b>	
<code>proc(i, 23);</code>	<b>✗</b>	Parámetro por referencia → ¡variable!
<code>proc(3.5, d);</code>	<b>✗</b>	¡Argumento double para parámetro int!



# Paso de argumentos. Ejemplo (I)

---

Subprograma para dividir dos números devolviendo su cociente y el resto

...

```
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    div = op1 / op2;  
    rem = op1 % op2;  
}  
  
int main() {  
    int cociente, resto;  
    for (int j = 1; j <= 4; j++) {  
        for (int i = 1; i <= 4; i++) {  
            divide(i, j, cociente, resto);  
            cout << i << " entre " << j << " da un cociente de "  
                << cociente << " y un resto de " << resto << endl;  
        }  
    }  
  
    return 0;  
}
```



# Paso de argumentos. Ejemplo (II)

---

```
...  
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    div = op1 / op2;  
    rem = op1 % op2;  
}
```

```
int main() {  
    int cociente, resto;  
    for (int j = 1; j <= 4; j++) {  
        for (int i = 1; i <= 4; i++) {  
            → divide(i, j, cociente, resto);  
            ...  
        }  
    }  
  
    return 0;  
}
```

Memoria	
cociente	?
resto	?
i	1
j	1
...	

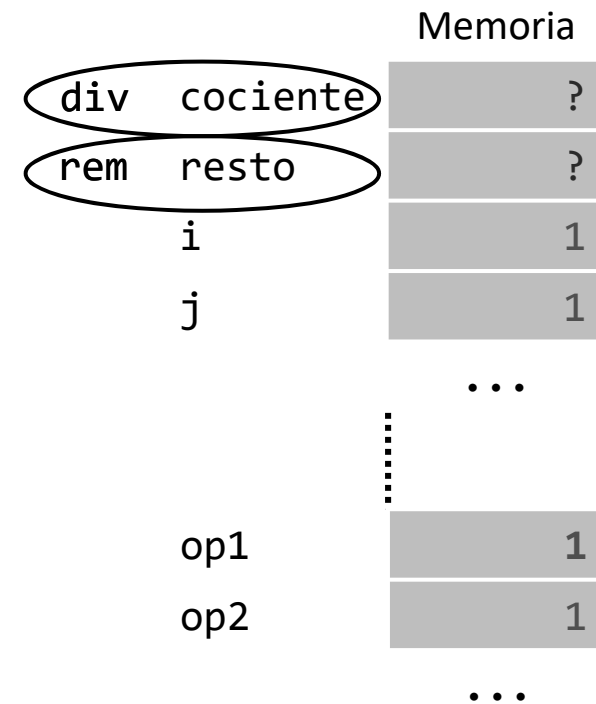


# Paso de argumentos. Ejemplo (III)

...

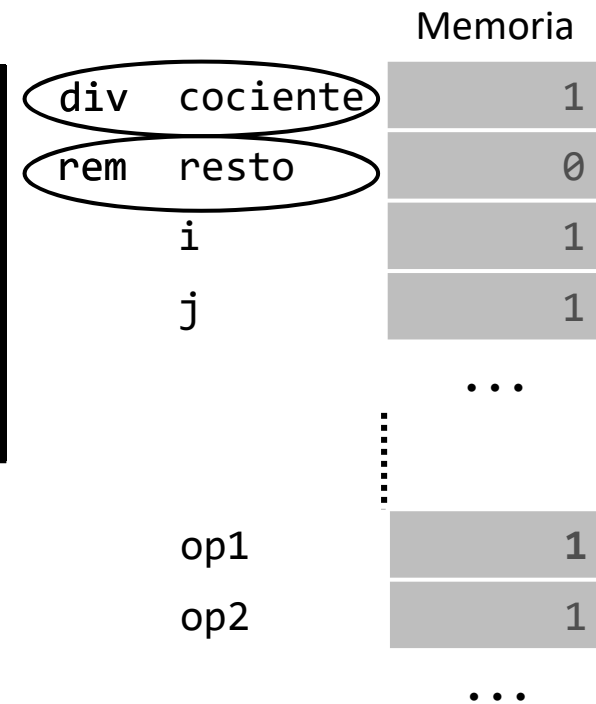
```
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    div = op1 / op2;  
    rem = op1 % op2;  
}
```

```
int main() {  
    int cociente, resto;  
    for (int j = 1; j <= 4; j++) {  
        for (int i = 1; i <= 4; i++) {  
            divide(i, j, cociente, resto);  
            ...  
        }  
    }  
  
    return 0;  
}
```



# Paso de argumentos. Ejemplo (IV)

```
...  
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    div = op1 / op2;  
    rem = op1 % op2;  
}  
  
int main() {  
    int cociente, resto;  
    for (int j = 1; j <= 4; j++) {  
        for (int i = 1; i <= 4; i++) {  
            divide(i, j, cociente, resto);  
            ...  
        }  
    }  
  
    return 0;  
}
```





# Paso de argumentos. Ejemplo (V)

---

```
...  
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    div = op1 / op2;  
    rem = op1 % op2;  
}
```

```
int main() {  
    int cociente, resto;  
    for (int j = 1; j <= 4; j++) {  
        for (int i = 1; i <= 4; i++) {  
            divide(i, j, cociente, resto);  
            ...  
        }  
    }  
  
    return 0;  
}
```

Memoria	
cociente	1
resto	0
i	1
j	1
...	



# Más ejemplos (I)

Subprograma para intercambiar el valor de dos variables

```
...  
void intercambia(double &valor1, double &valor2) {  
→ // Intercambia los valores  
    double tmp; // Variable local (temporal)  
    tmp = valor1;  
    valor1 = valor2;  
    valor2 = tmp;  
}
```

Memoria temporal  
del procedimiento

tmp	?
-----	---

...

```
int main() {  
    double num1, num2;  
    cout << "Valor 1: ";  
    cin >> num1;  
    cout << "Valor 2: ";  
    cin >> num2;  
    intercambia(num1, num2);  
    cout << "Ahora el valor 1 es " << num1  
        << " y el valor 2 es " << num2 << endl;  
    return 0;  
}
```

Memoria de main()

valor1	num1	13.6
valor2	num2	317.14

...



# Más ejemplos (II)

---

```
...
// Prototipo
void cambio(double precio, double pago, int &euros, int &cent50,
            int &cent20, int &cent10, int &cent5, int &cent2, int &cent1);

int main() {
    double precio, pago;
    int euros, cent50, cent20, cent10, cent5, cent2, cent1;
    cout << "Precio: ";
    cin >> precio;
    cout << "Pago: ";
    cin >> pago;
    cambio(precio, pago, euros, cent50, cent20, cent10, cent5, cent2,
           cent1);
    cout << "Cambio: " << euros << " euros, " << cent50 << " x 50c., "
         << cent20 << " x 20c., " << cent10 << " x 10c., "
         << cent5 << " x 5c., " << cent2 << " x 2c. y "
         << cent1 << " x 1c." << endl;

    return 0;
}
```



# Paso de parámetros. Resumen

	Ventajas	Desventajas
Por valor	<ul style="list-style-type: none"><li>• Aísla los parámetros evitando efectos colaterales en otros subprogramas</li><li>• Permite usar constantes o expresiones como argumento</li></ul>	<ul style="list-style-type: none"><li>• Utiliza más memoria, realiza copias de cada argumento</li></ul>
Por referencia	<ul style="list-style-type: none"><li>• Utiliza menos memoria, no hace copia</li></ul>	<ul style="list-style-type: none"><li>• Solo se pueden usar variables como argumento</li><li>• Puede producir efectos colaterales ya que los cambios realizados dentro del subprograma se propagan fuera</li></ul>



# Tipos de procedimientos

---

## 1. De entrada de datos:

- ✓ Solicitan datos a los usuarios y los cargan en variables
- ✓ Parámetros por referencia

## 2. De salida de datos:

- ✓ Muestran datos al usuario
- ✓ Parámetros por valor

## 3. Internos:

- ✓ No hacen E/S
- ✓ Reciben datos del programa y los modifican
- ✓ Parámetros por valor y por referencia



# Separación entre la lógica y la E/S

---

- ✓ Hay que separar los cálculos de la comunicación con el usuario
- ✓ Los subprogramas que realizan cálculos u operaciones no deberían pedir datos (`cin`) ni mostrar datos por consola (`cout`).
  - ✓ Devolverán el resultado de su operación
  - ✓ Será el subprograma que hace la llamada el que procesará los datos
- ✓ ¿Por qué? Para poder reutilizar subprogramas independientemente de:
  - Si la aplicación es de consola o tiene una interfaz gráfica de usuario
  - El idioma de la aplicación
  - Cómo se vayan a usar en el nuevo programa



# Notificación de errores (I)

---

En los subprogramas se pueden detectar errores que impiden realizar los cálculos:

```
void cambio(double precio, double pago, int &euros, int &cent50,  
            int &cent20, int &cent10, int &cent5, int &cent2, int &cent1) {  
→ { if (pago < precio) { // Cantidad insuficiente  
    cout << "Error: El pago es inferior al precio" << endl;  
    }  
    ...  
}
```

¿Debe el subprograma notificar al usuario o al programa?

→ Mejor notificarlo al punto de llamada y allí decidir qué hacer

```
void cambio(double precio, double pago, int &euros, int &cent50,  
            int &cent20, int &cent10, int &cent5, int &cent2, int &cent1,  
→ bool &error) {  
    if (pago < precio) { // Cantidad insuficiente  
→ error = true;  
    }  
    else {  
→ error = false;  
    }
```



# Notificación de errores (II)

---

Al volver de la llamada se decide qué hacer si ha habido error...

- ✓ ¿Informar al usuario?
- ✓ ¿Volver a pedir los datos?
- ✓ Etcétera

**cambio.cpp**

```
int main() {  
    double precio, pago;  
    int euros, cent50, cent20, cent10, cent5, cent2, cent1;  
    → bool error;  
    cout << "Precio: ";  
    cin >> precio;  
    cout << "Pago: ";  
    cin >> pago;  
    cambio(precio, pago, euros, cent50, cent20, cent10, cent5, cent2,  
           cent1, error);  
    → if (error) {  
        cout << "Error: El pago es inferior al precio" << endl;  
    }  
    else {  
        ...  
    }  
}
```





# Los subprogramas y el main. Prototipos

---

¿Dónde poner los subprogramas? ¿Antes o después del `main()`?

→ Los pondremos después de `main()`

Sin embargo, si ponemos los subprogramas después del `main`, el compilador no los conocerá cuando el `main` los invoque!!!

→ Pondremos el prototipo de los subprogramas antes de `main()`

→ El prototipo actúa como la “declaración” de una variable

**Prototipo:** cabecera del subprograma terminada en ;

```
void dibujarCirculo();  
void mostrarM();  
void proc(double &a);  
int cuad(int x);  
...
```



`main()` es el único subprograma que no hay que prototipar



# Llamando o invocando a un subprograma

---

¿Son correctas las llamadas a subprogramas?

- ¿Existe el subprograma?
  - ✓ El prototipo debe aparecer en el fichero antes de la invocación
  - ✓ El código del subprograma puede aparecer después o incluso estar en otro fichero
- ¿Concuerdan los argumentos con los parámetros?
  - ✓ *Los parámetros deben coincidir en tipo y orden*
  - ✓ *El prototipo, la definición de la función y las invocaciones deben concordar*



# Ejemplo Intercambio

intercambia.cpp

```
#include <iostream>
using namespace std;
```

```
void intercambia(double &valor1, double &valor2); // Prototipo
```

```
int main() {
    double num1, num2;
    cout << "Valor 1: ";
    cin >> num1;
    cout << "Valor 2: ";
    cin >> num2;
    intercambia(num1, num2);
    cout << "Ahora el valor 1 es " << num1
          << " y el valor 2 es " << num2 << endl;
    return 0;
}
```



Asegúrate de que los prototipos coincidan con las implementaciones

```
void intercambia(double &valor1, double &valor2) {
    double tmp; // Variable local (temporal)
    tmp = valor1;
    valor1 = valor2;
    valor2 = tmp;
}
```



# Ejemplo Mates

mates.cpp

```
#include <iostream>
using namespace std;

// Prototipos
long long int factorial(int n);
int sumatorio(int n);

int main() {
    int num;
    cout << "Num: ";
    cin >> num;
    cout << "Factorial de "
         << num << ": "
         << factorial(num)
         << endl
         << "Sumatorio de 1 a "
         << num << ": "
         << sumatorio(num)
         << endl;

    return 0;
}
```

```
long long int factorial(int n) {
    long long int fact = 1;

    if (n < 0) {
        fact = 0;
    }
    else {
        for (int i = 1; i <= n; i++) {
            fact = fact * i;
        }
    }

    return fact;
}

int sumatorio(int n) {
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }

    return sum;
}
```



# Comentarios en los prototipos

---

- ✓ Los prototipos deben ir comentados con información de:
  - ✓ Su funcionalidad
  - ✓ Los parámetros de entrada y salida, y qué significan

```
/* Divide op1 entre op2 y devuelve el cociente y el resto.  
   E: op1 (el dividendo) y op2 (el divisor)  
   S: div (el resultado de la división)  
       rem (el resto de la división)
```

```
*/  
void divide(int op1, int op2, int &div, int &rem);
```

```
/* Eleva un número entero al cuadrado.  
   E/S: x, el número que se quiere elevar al cuadrado. Sale  
   conteniendo el número inicial al cuadrado.
```

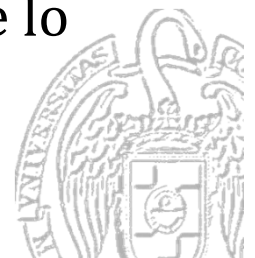
```
*/  
void cuadrado(int &x);
```



# Cuando usar subprogramas

---

- ✓ En el cuerpo del programa principal (main) se deben evitar la aparición excesiva de estructuras de control utilizando subprogramas
  - El cuerpo del programa principal debe contener, principalmente, llamadas a subprogramas
- ✓ Se deben de usar subprogramas para:
  - **Aumentar la legibilidad del código:** Si un algoritmo complejo se escribe sin subprogramas es difícil de entender. Dividiéndolo en subprogramas es más legible y podemos centrarnos en cada momento en entender problemas más pequeños
  - **Evitar la repetición de código:** Si hay un tratamiento que se hace varias veces a lo largo del programa debemos escribir un subprograma que haga este tratamiento y realizar una llamada a este subprograma cada vez que lo necesitemos



# Cuando usar subprogramas. Ejemplo1 (I)

```
#include <iostream>
using namespace std;
#include <cmath>

int main()
{
    double capital, interesAnual, ratio, cuota, totalPagado;
    int anios, plazo;
```

```
    cout << "Capital: ";
    cin >> capital;
    cout << "Interés anual: ";
    cin >> interesAnual;
    cout << "Años: ";
    cin >> anios;
```

Solicitud de datos

```
    ratio = interesAnual / 12;
    plazo = anios * 12;
    cuota = capital * ratio / (100 * (1 - pow((1 + ratio / 100), -plazo)));
```

Cálculos

```
    cout << "Cuota mensual: " << cuota << " €" << endl;
    totalPagado = cuota * plazo;
    cout << "Capital amortizado: " << capital << " €" << endl;
    cout << "Intereses: " << totalPagado - capital << " €" << endl;
```

Presentación de resultados

```
    return 0;
}
```



# Cuando usar subprogramas. Ejemplo1 (II)

```
#include <iostream>
using namespace std;
#include <cmath>

void solicitarDatos(double &capital, double &interesAnual, int &anios);
double calculoCuota(double interesAnual, int anios, double capital);
double calculoIntereses(double cuota, int anios, double capital);
void mostrarResultados(double cuota, double capital, double intereses);

int main()
{
    double capital, interesAnual, cuota, totalPagado, intereses;
    int anios;

    solicitarDatos(capital, interesAnual, anios);

    cuota = calculoCuota(interestAnual, anios, capital);
    intereses = calculoIntereses(cuota, anios, capital);

    mostrarResultados(cuota, capital, intereses);
    system("pause");
    return 0;
}

void solicitarDatos(double &capital, double &interesAnual, int &anios){
    cout << "Capital: ";
    cin >> capital;
    cout << "Interés anual: ";
    cin >> interesAnual;
    cout << "Años: ";
    cin >> anios;
}

double calculoCuota(double interesAnual, int anios, double capital){
    double ratio, cuota;
    int plazo;

    ratio = interesAnual / 12;
    plazo = anios * 12;
    cuota = capital * ratio / (100 * (1 - pow((1 + ratio / 100), -plazo)));

    return cuota;
}

double calculoIntereses(double cuota, int anios, double capital){
    double intereses, totalPagado;
    int plazo = anios * 12;;

    totalPagado = cuota * plazo;
    intereses = totalPagado - capital;

    return intereses;
}

void mostrarResultados(double cuota, double capital, double intereses){
    cout << "Cuota mensual: " << cuota << " €" << endl;
    cout << "Capital amortizado: " << capital << " €" << endl;
    cout << "Intereses: " << intereses << " €" << endl;
}
```

¡Más legible!





# Cuando usar subprogramas. Ejemplo2 (I)

```
#include <iostream>
using namespace std;
```

```
int main(){
    int opcion;
```

```
    cout << "Elija la opcion que desea realizar:" << endl;
    cout << "1.- Comprobar si un numero positivo es primo." << endl;
    cout << "2.- Comprobar is un numero positivo es par." << endl;
    cout << "0.- Salir" << endl;
    cin >> opcion;
```

menu

```
    switch(opcion){
    case 1:{
```

```
        int numero;
        cout << "Introduzca un número positivo: ";
        cin >> numero;
        if (numero > 0){
            bool esPrimo = true;
            int contador = 2;
            while (esPrimo && contador <= numero/2){
                if (numero%contador == 0)
                    esPrimo = false;
                else
                    contador++;
            }
            if (esPrimo)
                cout << "El número introducido es primo" << endl;
            else
                cout << "El número introducido no es primo" << endl;
        }
    }
```

comprobarPrimo

```
        break;
    case 2:{
```

```
        int numero;
        cout << "Introduzca un número positivo: ";
        cin >> numero;
        if (numero > 0){
            if (numero%2 == 0)
                cout << "El número introducido es par" << endl;
            else
                cout << "El número introducido no es par" << endl;
        }
    }
```

comprobarPar

```
        break;
    }
    return 0;
}
```



# Cuando usar subprogramas. Ejemplo2 (II)

```
#include <iostream>
using namespace std;

int menu();
void comprobarPrimo();
void comprobarPar();
```

```
int main(){
    int opcion;

    opcion = menu();

    switch(opcion){
        case 1: comprobarPrimo();
                break;
        case 2: comprobarPar();
                break;
    }

    return 0;
}
```

```
int menu(){
    int opcion;

    cout << "Elija la opcion que desea realizar:" << endl;
    cout << "1.- Comprobar si un numero positivo es primo." << endl;
    cout << "2.- Comprobar si un numero positivo es par." << endl;
    cout << "0.- Salir" << endl;
    cin >> opcion;

    return opcion;
}
```

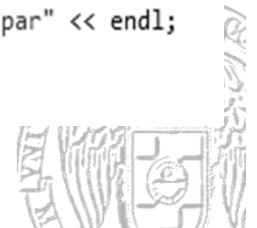
¡Más legible!

```
void comprobarPrimo(){
    int numero;
    cout << "Introduzca un numero positivo: ";
    cin >> numero;

    if (numero > 0){
        bool esPrimo = true;
        int contador = 2;
        while (esPrimo && contador <= numero/2){
            if (numero%contador == 0)
                esPrimo = false;
            else
                contador++;
        }
        if (esPrimo)
            cout << "El numero introducido es primo" << endl;
        else
            cout << "El numero introducido no es primo" << endl;
    }
}
```

```
void comprobarPar(){
    int numero;
    cout << "Introduzca un numero positivo: ";
    cin >> numero;

    if (numero > 0){
        if (numero%2 == 0)
            cout << "El numero introducido es par" << endl;
        else
            cout << "El numero introducido no es par" << endl;
    }
}
```



# Cuando usar subprogramas. Ejemplo2 (III)

```
void comprobarPrimo(){
    int numero = solicitarNumero();
    if (numero > 0){
        bool esPrimo = true;
        int contador = 2;
        while (esPrimo && contador <= numero/2){
            if (numero%contador == 0)
                esPrimo = false;
            else
                contador++;
        }
        if (esPrimo)
            cout << "El numero introducido es primo" << endl;
        else
            cout << "El numero introducido no es primo" << endl;
    }
}

void comprobarPar(){
    int numero = solicitarNumero();
    if (numero > 0){
        if (numero%2 == 0)
            cout << "El numero introducido es par" << endl;
        else
            cout << "El numero introducido no es par" << endl;
    }
}

int solicitarNumero(){
    int numero;

    cout << "Introduzca un numero positivo: ";
    cin >> numero;

    return numero;
}
```



# Refinamientos sucesivos (I)

---

## Paso 0.- Especificación inicial

*Desarrollar un programa que haga operaciones de conversión de medidas hasta que el usuario decida que no quiere hacer más*

- ✓ Análisis y diseño aumentando el nivel de detalle en cada paso  
*¿Qué operaciones de conversión?*

## Paso 1.-

*Desarrollar un programa que haga operaciones de conversión de medidas hasta que el usuario decida que no quiere hacer más*

- ★ *Pulgadas a centímetros*
- ★ *Libras a gramos*
- ★ *Grados Fahrenheit a centígrados*
- ★ *Galones a litros*



# Refinamientos sucesivos (II)

---

*Paso 2.-*

*Desarrollar un programa que muestre al usuario un menú con cuatro operaciones de conversión de medidas:*

- ✱ Pulgadas a centímetros*
- ✱ Libras a gramos*
- ✱ Grados Fahrenheit a centígrados*
- ✱ Galones a litros*

*Y lea la elección del usuario y proceda con la conversión, hasta que el usuario decida que no quiere hacer más*

6 grandes tareas:

Menú, cuatro funciones de conversión y `main()`



# Refinamientos sucesivos (III)

---

Paso 2.-



# Refinamientos sucesivos (IV)

---

Paso 3.-

★ *Menú:*

Mostrar las cuatro opciones más una para salir  
Validar la entrada y devolver la elegida

★ *Pulgadas a centímetros:* Devolver el equivalente en centímetros del valor en pulgadas

★ *Libras a gramos:* Devolver el equivalente en gramos del valor en libras

★ *Grados Fahrenheit a centígrados:* Devolver el equivalente en centígrados del valor en Fahrenheit

★ *Galones a litros:* Devolver el equivalente en litros del valor en galones

★ *Programa principal (main())*



# Refinamientos sucesivos (V)

---

Paso 3.- Cada tarea, un subprograma

Comunicación entre los subprogramas:

Función	Entrada	Salida	Valor devuelto
menu()	—	—	int
pulgACm()	double pulg	—	double
lbAGr()	double libras	—	double
grFAGrC()	double grF	—	double
galALtr()	double galones	—	double
main()	—	—	int





# Refinamientos sucesivos (VI)

---

Paso 4.- Algoritmos detallados de cada subprograma → Programar

```
#include <iostream>
using namespace std;
// Prototipos
int menu();
double pulgACm(double pulg);
double lbAGr(double libras);
double grFAGrC(double grF);
double galALtr(double galones);

int main() {
    double valor;
    int op = -1;
    while (op != 0) {
        op = menu();
        switch (op) {
            case 1:
                {
                    cout << "Pulgadas: ";
                    cin >> valor;
                    cout << "Son " << pulgACm(valor) << " cm." << endl;
                }
                break;
            ...
        }
    }
}
```



# Refinamientos sucesivos (VII)

---

```
    case 2:
    {
        cout << "Libras: ";
        cin >> valor;
        cout << "Son " << lbAGr(valor) << " gr." << endl;
    }
    break;
    case 3:
    {
        cout << "Grados Fahrenheit: ";
        cin >> valor;
        cout << "Son " << grFAGrC(valor) << " °C" << endl;
    }
    break;
    case 4:
    {
        cout << "Galones: ";
        cin >> valor;
        cout << "Son " << galALtr(valor) << " l." << endl;
    }
    break;
}
return 0;
}
```

...



# Refinamientos sucesivos (VIII)

---

```
int menu() {
    int op = -1;

    while ((op < 0) || (op > 4)) {
        cout << "1 - Pulgadas a Cm." << endl;
        cout << "2 - Libras a Gr." << endl;
        cout << "3 - Fahrenheit a °C" << endl;
        cout << "4 - Galones a L." << endl;
        cout << "0 - Salir" << endl;
        cout << "Elige: ";
        cin >> op;
        if ((op < 0) || (op > 4)) {
            cout << "Opción no válida" << endl;
        }
    }

    return op;
}

double pulgACm(double pulg) {
    const double cmPorPulg = 2.54;
    return pulg * cmPorPulg;
}
```

...



# Refinamientos sucesivos (IX)

conversiones.cpp

```
double lbAGr(double libras) {  
    const double grPorLb = 453.6;  
    return libras * grPorLb;  
}
```

```
double grFAGrC(double grF) {  
    return ((grF - 32) * 5 / 9);  
}
```

```
double galALtr(double galones) {  
    const double ltrPorGal = 4.54609;  
    return galones * ltrPorGal;  
}
```



# Diseño descendente. Ventajas

---

- ✓ Descompone la complejidad del problema
- ✓ Es más sencillo ya que se puede aplazar el desarrollo de algunas subtarear
- ✓ Pueden programar independientemente los miembros del equipo
- ✓ Es mucho más fácil encontrar cosas en el código, sobre todo si es un programa largo
- ✓ El mantenimiento es más sencillo ya que se modifican subprogramas independientes del resto del programa
- ✓ Permite usar subprogramas usados por otros programadores
- ✓ Evita cambios indeseables en las variables del programa. Tan solo las variables de entrada/salida se van transfiriendo entre programa y subprogramas






# Acerca de *Creative Commons*



## *Licencia CC (Creative Commons)*

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Material original elaborado por Luis Hernández Yáñez, con modificaciones de Raquel Hervás Ballesteros y Virginia Francisco Gilmartín

